1

State of the Art on Network Troubleshooting

"A protocol approach to troubleshooting"

Ed Wilson

Chapter 1 presents the state of the art on network troubleshooting and a traditional troubleshooting architecture for non-encrypted traffic. We then discuss its limitations when traffic is encrypted.

1.1. Network troubleshooting

In the early 19th century, technicians were dispatched to find problems in telegraph and phone line infrastructure to repair and solve the issues. Historically, a troubleshooter refers to a skilled worker who finds and solves technical problems. Nowadays, *troubleshooting* is a form of problem-solving that aims to repair failed processes in a machine or a system. According to the related work Morris and Rouse (1985) and Jonassen and Hung (2006), there are several existing conceptions of the troubleshooting process. The basic concept of troubleshooting is finding the faulty components in a device to repair or replace it Perez (1991). Schaafstal et al. (2000) designed the troubleshooting process with four subtasks: formulating problem description, cause generation, test and evaluation. Similarly, troubleshooting is considered as an iterative process with four subprocesses: problem space construction,

problem space reduction, fault diagnosis and solution verification (Johnson et al. 1993).

Network troubleshooting is an iterative process with three subtasks: identifying, diagnosing and solving problems in the network. In the past, network operators (NOs) implemented manual troubleshooting tools such as *ping*, *traceroute*, etc. *ping* is a computer network administration utility designed to check a reachability between a source and a destination and round-trip time of packets in the network. *traceroute* is a computer network diagnostic utility used to display possible routes between a source and a destination and measure a transit delay of packets in the network. These troubleshooting tools are used to diagnose complex problems such as loops caused by undefined interaction between spanning tree protocols (Heller et al. 2013), etc. However, these approaches are not effective with a huge number of network devices. Besides, 24.6% of administrators reported that anomaly diagnosis takes more than 1 h on average to solve anomalies (Zeng et al. 2012a). Therefore, it is necessary for an automated troubleshooting process that aims to detect an anomaly, locate its causes and solve it. Consequently, network troubleshooting is considered by the research community Fonseca and Mota (2017); Yu et al. (2018); Cherrared et al. (2019). In the following section, we present the state of the art of network troubleshooting.

1.1.1. *State of the art*

According to the related work on network troubleshooting (Yu et al. 2018; Fonseca and Mota 2017; Van et al. 2018), problems can be classified into several categories thanks to locations where problems happen or factors that result in problems. Yu et al. (2018) and Fonseca and Mota (2017) categorize problems into problems in application, control and infrastructure layer. Similarly, problems can be classified into problems in application service providers (ASP) or Internet service providers (ISP) (Van et al. 2018). Besides, problems can be classified into problems caused by administrators (e.g. router misconfiguration, server misconfiguration, etc.) or problems that are not caused by administrators (e.g. link failure, switch failure, buffer overload, etc.). According to a survey of NOs (Zeng et al. 2012b), in this book, we present several problems that are not caused by administrators in following sections.

1.1.1.1. *Rule failure*

Bu et al. (2016) categorized failure rule in the network into missing fault and priority fault. The missing fault occurs when a rule is not executed as expected, whereas the priority fault occurs when overlapping rules violate a priority order.

There are research studies concentrating on the missing fault including ATPG in Zeng et al. (2012a) and Monocle Perešíni et al. (2015). These approaches verify the rules by generating probe packets to exercise every rule. ATPG uses a header space analysis (Kazemian et al. 2012) to check the reachability between all test hosts. Then, the reachability result is transferred to a probe packet generator to compute a minimal set of probe packets via greedy algorithm (Slavık 1997). Next, these probe packets are sent into the network systems to check the rule's corrections. If an error is detected, a fault localization algorithm is implemented to narrow down to identify the root cause. However, ATPG has a drawback when it generates the probe packets for all rules. It is not effective when there are only a few up-to-date rules. Consequently, Monocle is proposed to overcome this drawback. This approach only verifies recently installed rules and reports misbehaviors. Besides, Monocle formulates knowledge from flow tables in the switches as constraints and applies an SAT solver (Biere 2008) to generate a set of probe packets.

Probing is an intrusive method that generates significant overheads and increases link utilization in the network. Consequently, it is necessary to minimize the number of probe packets. This is a minimum set cover problem, which is an NP-Complete problem (Zeng et al. 2012a). Therefore, Bu et al. (2016) proposed RuleScope, a framework for detecting rule failures in the network. RuleScope divides flow tables into solvable subsets of rules to minimize probe scale. Then, this approach creates a directed acyclic graph for each subset and generates a set of probe packets for each subset. As a result, this approach processes the probe packet generation more quickly due to a small scale of rule subsets.

Although RuleScope minimizes the number of probe packets, this approach suffers from a drawback related to a separation in the flow tables. This leads to the priority fault in the switches. The separation in the flow tables into small subsets can result in pretermitting two overlapping rules in two different subsets of rules. Zhao et al. (2018a) proposed SERVE, a rule

verification to identify rule failure in the switches automatically. Firstly, SERVE extracts all rules for each device and builds a multi-rooted tree that considers rule connections. Next, SERVE analyzes the multi-rooted tree to generate the minimum number of probe packets. The minimum set cover problem is an NP-Complete problem, so SERVE applies the depth-first search (DFS) algorithm to generate the probe packets. Zhao et al. (2018b) extended the previous study of Zhao et al. (2018a) to present a complete framework. After generating the probe packets, SERVE injects these packets into network systems using an out-band channel. Besides, SERVE also computes a desired network behavior using the multi-rooted trees. According to a comparison between the feedback from the out-band channel for every rule and the desired network behaviors, SERVE can detect faulty rules and send notifications to administrators. SERVE's performance is evaluated to benchmarks in processing time, number of probe packets and overheads. Concerning the number of probe packets, SERVE decreases the number of probe packets by up to 75% in comparison with Monocle. Regarding the processing time, SERVE's figure is three times less than the figure for ATPG. As for the overhead, in-band bandwidth is not influenced according to using the out-band channel to inject the probe packets. Besides, the out-band bandwidth is far less than link capacity.

1.1.1.2. *Link failure*

Link failure refers to unreachability between two switches. It can lead to a high packet loss and performance degradation in the network. Link failure can be detected according to probe packets in active monitoring approaches. *ping* is a simple troubleshooting tool that sends probe packets to check the reachability between two end-points. If probe packets are lost, it means that there is a faulty link between these end-points. Similarly, Cascone et al. (2017) proposed a fast failure detection mechanism to detect the link failure based on the exchange of bidirectional "heartbeat" packets. When the packet rate drops below a threshold, a node sends heartbeat packets to its neighbors. If there are no responses from its neighbors after a given time, the link failure happens in the network. However, this mechanism requires a strict consumption related to the backup solutions that cannot be utilized to guarantee the short failover delays (1 ms).

Moreover, this problem can be detected by using the Link Layer Discovery Protocol (LLDP) in software-defined networking (SDN) (Khan

et al. 2016; Tarnaras et al. 2015). According to the topology discovery protocol, SDN controller can detect link failure and remove it from network topology. Firstly, an OpenFlow (OF) switch connects to the controller so that the controller knows its active ports. Next, the controller generates a Packet-out message to each active port in the switch to discover the topology. The LLDP between switch s_1 and s_2 is depicted in Figure 1.1. Firstly, the controller encapsulates an LLDP packet in a Packet-out message and sends it to the switch s_1. When switch s_1 receives the Packet-out message, it will forward the LLDP packet to switch s_2. After receiving the LLDP packet, switch s_2 encapsulates this packet in a Packet-in message and sends it back to the controller. The controller receives this message and creates a link from switch s_1 to s_2. The same process is performed to identify the link for an opposite direction. When link s_1–s_2 is faulty, the controller will not receive the Packet-in message from switch s_2. Then, the controller will remove this link from the network topology. In the network with S switches interconnected by a set of L links, the total number of Packet-out and Packet-in messages are described in equations [1.1] and [1.2], respectively. P_i is the number of the active port in the switch S_i.

$$TOTAL_{PACKET-OUT} = \sum_{i=1}^{S} P_i \quad [1.1]$$

$$TOTAL_{PACKET-IN} = 2L \quad [1.2]$$

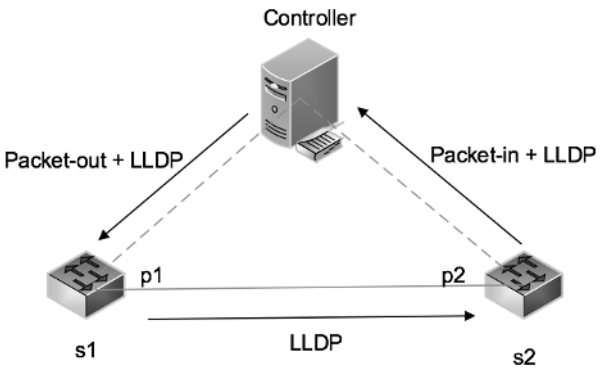

Figure 1.1. *Unidirectional link discovery in LLDP. For a color version of this figure, see www.iste.co.uk/tong/troubleshooting.zip*

Unlike the SDN environments, a hybrid SDN contains OF switches and traditional switches that LLDP cannot discover. Therefore, SDN controllers

(e.g. Floodlight Floodlight (n.d.), OpenDayLight **?**, etc.) utilize a combination of LLDP and Broadcast Domain Discovery Protocol (BDDP) (Khan et al. 2016) for the topology discovery. There are two main differences between LLDP and BDDP. First of all, destination MAC address in BDDP is a broadcast address (FF:FF:FF:FF:FF:FF) in contrast to multicast addresses used by LLDP. This allows the traditional switch forwarding the BDDP packets to detect multi-hop links, whereas LLDP is a mechanism for a single-hop link detection. Secondly, EtherType of LDDP is 0x88cc while EtherType of BDDP is 0x8999.

To keep up-to-date topology information, the controller needs to send $TOTAL_{PACKET-OUT}$ Packet-out messages and $TOTAL_{PACKET-IN}$ Packet-in messages periodically. This leads to a significant control overhead and an increase in link utilization. Therefore, many studies focus on optimizing the number of control packets for the topology discovery. Pakzad et al. (2014) proposed OpenFlow Discovery Protocol version 2 (OFDPv2) to reduce the number of Packet-out messages. Firstly, OFDPv2 reduces the number of LLDP Packet-out messages sent to each switch to one. Then, this protocol installs a new rule in each switch to forward LLDP packets received from the controller to all available ports. Finally, OFDPv2 customizes the event handler of the Packet-in message in the controller to parse the MAC address.

Besides, Xu et al. (2017) proposed a monitoring link failure detection approach to reduce the number of Packet-out messages in LLDP to one. Firstly, this approach calculates a monitoring tree path based on network topology. Then, this approach installs monitoring flow entries into each switch based on the switch's locations in the monitoring tree path to match and apply actions to monitoring packets. Next, the controller sends a Packet-out message to a source node to instruct this node to send out the monitoring packets to other nodes following the monitoring tree. If a switch cannot receive the LLDP packet after a given time, this switch will send an alarm packet to the controller.

Unlike sending LLDP packets periodically to network systems to update the topology information, Azzouni et al. (2018) proposed sOTDP, a secure and efficient OpenFlow Discovery Protocol that applies bidirectional forwarding detection (BFD) (Katz and Ward 2010) to detect link's events quickly. The controller listens to link's event notifications from switches to

update the network topology instead of updating it periodically by sending LLDP packets. BFD is a detection protocol designed to provide a fast failure detection of forwarding paths. This protocol establishes a session between two endpoints over a particular link and sends control messages to detect active links.

1.1.1.3. *Buffer overload*

Switch's buffer overload occurs when a data volume exceeds the storage capacity of buffer memory in switches. This can lead to congestion in the network. Phanishayee et al. (2008) highlighted that an increase in buffer size can solve the buffer overload due to a massive amount of network traffic. Nevertheless, this approach has low scalability because the replacement cost of conventional devices is extremely high. Therefore, many studies focus on congestion control and congestion avoidance mechanisms to prevent and avoid the buffer overload.

Concerning the congestion control mechanisms, there are two kinds of flows in the network: elephant flows and mice flows. Elephant flows are the large continuous flows that consume high bandwidth, whereas mice flows are the small flows that consume low bandwidth. The elephant flows contain more than 90% of all bytes transmitted in the network (Kandula et al. 2009; Benson et al. 2010). Thus, Kanagevlu and Aung (2015) proposed a local rerouting mechanism for elephant flows in the SDN environment to decrease negative impacts of buffer overload. If link utilization is higher than 75% of link capacity, an alarm is triggered. The controller reroutes the elephant flows into other paths with minimal link utilization to avoid the congested links. As for the mice flows, its lifetime is short, but the percentage of mice flows in network traffic is very high, approximately 90% (Kandula et al. 2009; Benson et al. 2010). Trestian et al. (2013) proposed MiceTrap, an OpenFlow-based traffic engineering approach for mice flows. Firstly, the network traffic is classified into elephant flows and mice flows. The elephant flows are processed as in previous research studies (e.g. Kanagevlu and Aung (2015), etc.), whereas the mice flows are processed by MiceTrap. If a traffic volume exceeds a threshold, MiceTrap is triggered. Then, MiceTrap routes the mice flows using a weighted multi-path routing algorithm, which assigns these flows into paths based on its weight. The higher the path's weight, the less the path is chosen.

Besides, there are studies focusing on both elephant and mice flows in the routing approach. Song et al. (2016) proposed a routing algorithm which reroutes congested links when link utilization is higher than 70% of link capacity. However, the routing algorithms without considering available bandwidth can lead to congested links in the network. Thus, Attarha et al. (2017) and Gholami and Akbari (2015) proposed a routing algorithm that estimates link utilization and selects the links with high available bandwidth. Similarly, Sminesh et al. (2018) proposed a routing algorithm using a Bayesian network. This approach contains three modules. The first one is to identify a bottleneck link when its link utilization is higher than a threshold. The second one is to update topology and identify all available routing paths. The final one is to select the routing paths using the Bayesian network according to link utilization and residual bandwidth.

The above routing mechanisms consider a routing policy for different applications (e.g. video streaming, file transfer, etc.). However, it is ineffective because each application contains specific SLA requirements (e.g. low latency, high bandwidth, etc.). Consequently, many research work placed a special focus on application-aware routing algorithms. Li et al. (2016) proposed an application-aware traffic control scheme that implements a simple flow classification to apply corresponding rules. Adami et al. (2015) proposed a differentiated routing algorithm in the network. Firstly, this approach applies a deep packet inspection to identify the Session Initiation Protocol (SIP) flows using its signatures (e.g. UDP packet with destination port 5060, content-Type is set to application/sdp, etc.). Next, this approach calculates the shortest paths for these flows using the Dijkstra algorithm with link utilization as a link cost. Similarly, Cheng et al. (2015) proposed an application-aware routing algorithm to implement different routing policies corresponding to various applications. Firstly, this method classifies the flows into three categories consisting of real-time applications (VoIP, video communication and gaming), streaming applications (streaming video, audio, IPTV and web browsing) and miscellaneous applications (file sharing and miscellaneous upload/download). Then, each category is processed with a specific routing policy using different network parameters such as link load, delay and delay variation. Thomas et al. (2016) proposed AMPF, an application-aware multi-path packet forwarding framework in SDN using a machine learning algorithm. First, AMPF classifies network flows into real-time traffic, buffered transfer, web browsing and restricted transfer. Next,

AMPF calculates k available paths using Dijkstra's algorithm corresponding to each application. Finally, AMPF assigns network flows into these routing paths.

Regarding the congestion avoidance algorithms, Abdelmoniem and Bensaou (2015) proposed a switch-assisted TCP congestion control approach using a small modification to switches. When congestion is detected, this approach rewrites the receiver window field in TCP header to reduce sending rate. Similarly, Hwang et al. (2015) proposed SCCP, a scalable congestion control protocol. SCCP limits sending rate of TCP senders by leveraging switches. Concretely, SCCP modifies the advertisement window field in the TCP header of packets traversing the switches. Besides, many studies are focusing on end-to-end congestion avoidance algorithms which adjust sending rate at the sender's side (Zhang et al. 2019; Zhang and Mao 2020). Xu et al. (2019) proposed DRL-CC, a congestion control algorithm using deep reinforcement learning (DRL) in MPTCP. DRL-CC monitors acknowledgment packets to extract network parameters (e.g. RTT, RTT deviation, goodput, etc.) and implements a DRL algorithm to adjust the congestion window (cwnd) at the sender's side. Similarly, Jay et al. (2019) proposed Auro, an Internet congestion control algorithm using DRL. Auro monitors network states and adjusts sending rate using the DRL algorithm and network parameters including latency gradient, latency ratio and sending ratio.

1.1.2. *Traditional troubleshooting architecture*

Following existing survey on network troubleshooting (Fonseca and Mota 2017; Yu et al. 2018; Cherrared et al. 2019), the traditional troubleshooting architecture contains four main modules including data collection, anomaly detection, root cause analysis and remediation modules. The traditional troubleshooting architecture is depicted as in Figure 1.2. Firstly, the data collection module monitors the network systems and collects the network traffic using monitoring approaches (e.g. NetFlow (Suárez-Varela and Barlet-Ros 2017), sFlow (Ujjan et al. 2020), OpenFlow-based monitoring approaches, etc.) to extract troubleshooting data (e.g. network parameters, etc.). Then, these data are analyzed in the anomaly detection module to detect anomalies (moments when network problems happen) in the network. After that, the root cause analysis analyzes thoroughly the troubleshooting data to

identify the root cause of anomalies in the network. An anomaly can be caused by many root causes. For instance, congestion can result from link failure, DDoS attack, switch's buffer overload, etc. Finally, root cause is solved definitively in the remediation module to return normal states to the network. The purpose of the troubleshooting architecture is to build self-healing networks that automatically detect, diagnose and remediate the network anomalies.

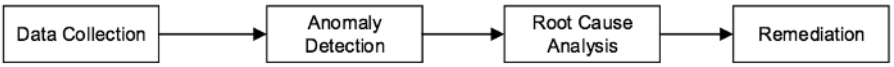

Figure 1.2. *Overall traditional troubleshooting architecture*

Facing a growing number of data stealing attacks, encrypted traffic is utilized by many service providers to make their data more secure and protect their user's privacy. This leads to many difficulties for network troubleshooting (Kühlewind et al. 2018). In the following sections, we present how network traffic is encrypted and drawbacks of network troubleshooting in the context of encrypted traffic.

1.2. Background on encryption protocols

In the report of Thales e-Security in April 2017 e-Security (n.d.), the percentage of encrypted flows ranges from 16% to 41% during the period 2005–2016. Figure 1.3 shows the global growth of encrypted traffic from 2005 to 2016. Similarly, in a current Cisco report Cisco (2021a), 80% of web traffic is encrypted by 2019 in comparison with 40% in 2016.

According to the related work of Velan et al. (2015), there are two kinds of encryption protocols in the network. The first one encrypts the whole packet (e.g. IPsec, etc.) while the second one only encrypts packets' payload (e.g. TLS, etc.). Besides, Google has recently developed Quick UDP Internet Connections (QUIC), a new transport layer network protocol from 2012, to provide secure connections and improve latency in the network (Langley et al. 2017). Similar to Transport Layer Security (TLS), QUIC only encrypts the payload and a part of packet's header. QUIC is equivalent to a combination of TCP and TLS. Moreover, QUIC offers opportunities to overcome the limitations of TCP + TLS related to connection establishment

time, TCP Head of Line blocking and connection migration (Langley et al. 2017). Therefore, in this book, we consider QUIC as an encryption protocol in the network troubleshooting framework. The background about IPsec, TLS and QUIC are presented as follows.

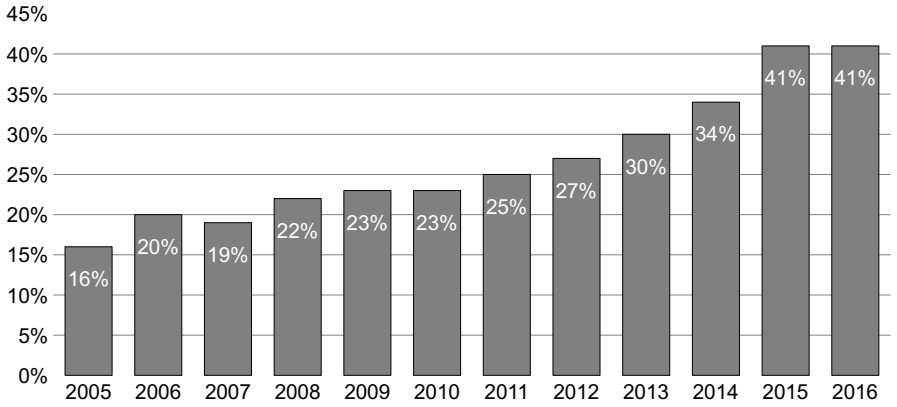

Figure 1.3. *The global growth of the encrypted traffic e-Security (n.d.). For a color version of this figure, see www.iste.co.uk/tong/troubleshooting.zip*

1.2.1. *QUIC*

QUIC (Langley et al. 2017) is a novel transport protocol designed by Google from 2012. Its design is motivated by multiple existing protocols such as TCP, TLS and HTTP/2 (Hypertext Transfer Protocol version 2). In early 2021, QUIC was standardized by Internet Engineering Task Force (IETF) to solve urgent issues of TCP + TLS (Kosek et al. 2021). The difference between TCP + TLS and QUIC is depicted as in Figure 1.4. QUIC is designed to improve latency in the network by sending data directly with 0-RTT in the best case. Moreover, it incorporates TLS1.3 to require all connections to be encrypted. QUIC also offers multiplexing features to send multiple data streams over a single connection. Furthermore, it offers opportunities to migrate connections without interrupting data communication. QUIC is implemented on the top of User Datagram Protocol (UDP), so it can be deployed in user space in contrast to TCP, which is implemented in system kernels. This feature allows us to deploy QUIC quickly in an application

update cycle. The deep analysis on QUIC's characteristics (Cui et al. 2017) is described as follows.

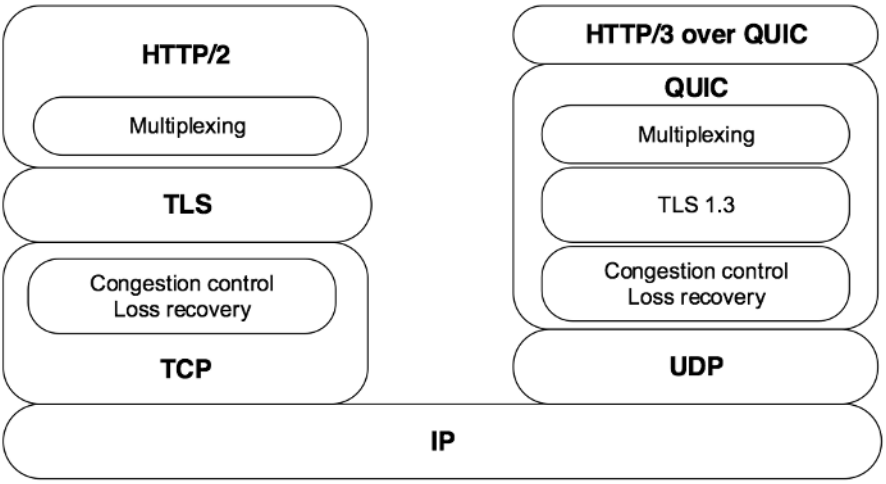

Figure 1.4. *Difference between TCP + TLS and QUIC architecture*

1.2.1.1. *Encryption*

QUIC uses TLS1.3 to provide end-to-end encryption. When the TLS handshake is complete, the encryption is performed on the UDP payload. In QUIC protocol, there are two kinds of headers: public and private headers. The public header is not encrypted, whereas QUIC encrypts its private header along with the payload. The comparison of QUIC packet format with TCP + TLS is depicted as in Figure 1.5. QUIC public header contains connection ID and flags. The connection ID is an identifier used to identify a connection at an endpoint. The flags are used in the initial session establishment.

The remainder (private header and payload) is encrypted, so it is not visible to an eavesdropper. In TCP, the control of the protocol is explicit, so a third party (e.g. NOs, etc.) is able to inspect this information (e.g. sequence number, acknowledge number, etc.) for network troubleshooting. For example, NOs can infer network performance parameters (e.g. round-trip time, etc.) by inspecting TCP packets. In contrast, the private header and payload of QUIC are encrypted to protect from third-party inspection. This

creates many obstacles for NOs in network troubleshooting. The detail is described thoroughly in section 1.3.

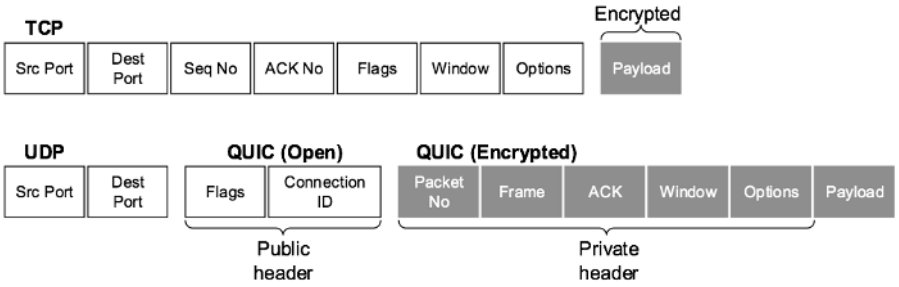

Figure 1.5. *Comparison of QUIC packet format with TCP + TLS. For a color version of this figure, see www.iste.co.uk/tong/troubleshooting.zip*

1.2.1.2. *Connection establishment*

Nowadays, many services require a secure and reliable connection. The combination of TCP and TLS is a potential solution for this purpose. TCP + TLS requires at least two RTTs to establish a secure connection, bringing a significant latency. QUIC integrates TLS1.3 to set up a secure connection with 0-RTT for a connection establishment time. In other words, encrypted payload is sent in the first packet when a previous connection is resumed. Comparison of connection establishment between QUIC and TCP + TLS is described as in Figure 1.6.

Concerning first-time connection establishment (Figure 1.6(a)), QUIC only uses 1-RTT by combining transport and cryptographic handshake in contrast to 3-RTTs in TCP + TLS1.2 and 2-RTTs in TCP + TLS1.3. QUIC combines both transport and cryptographic handshake in the first packet of connection to reduce latency in connection establishment. When a client connects to a server for the first time, it sends a Client Hello message to the server for key negotiation, along with other information (e.g. connection ID, preferred version number, etc.). The client encodes the handshake using the version number which is proposed. If the server does not support this version, it redirects the client to a version negotiation process. Otherwise, the server replies with a Server Hello message containing necessary information (e.g. certificate, session information, etc.) for subsequent connections. Next, the client can send encrypted packets to the server.

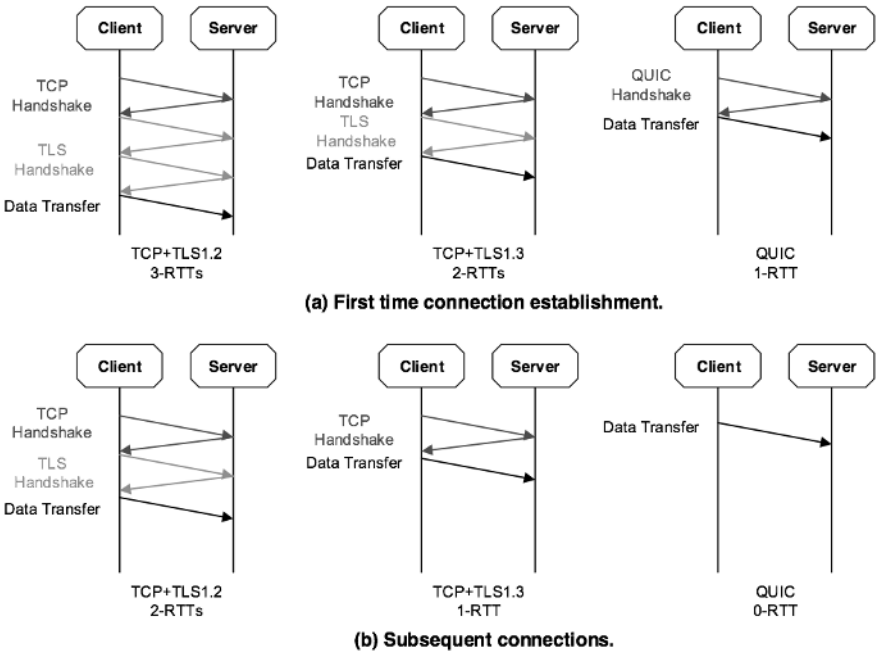

Figure 1.6. *Comparison of connection establishment between QUIC and TCP + TLS. For a color version of this figure, see www.iste.co.uk/tong/troubleshooting.zip*

The necessary information for the first-time connection establishment is stored in a cryptographic cookie at the client-side. It is used to authenticate the client in the subsequent connections. In the cookie, the server's Diffie–Hellman value is used to calculate an encryption key.

As for the subsequent connections (Figure 1.6(b)), the client sends the Diffie–Hellman value and its cookie to the server, along with the encrypted payload. The server uses information in the cookie to authenticate the client. Then, it uses the Diffie–Hellman value to calculate the encryption key to decrypt the encrypted payload and sends back the encrypted responses to the client.

1.2.1.3. *Multiplexing*

Multiplexing is a method of sending multiple data streams over a single connection. In HTTP1.1, a client can only request one resource in a single

connection as in Figure 1.7(a), so it needs to open multiple concurrent TCP connections. However, each client has a limited number of connections to a server, so sending a new request over one of these connections has to wait until a previous connection is complete. This leads to the HTTP Head of Line blocking (HoL blocking), which brings additional latency and complexity of managing multiple connections.

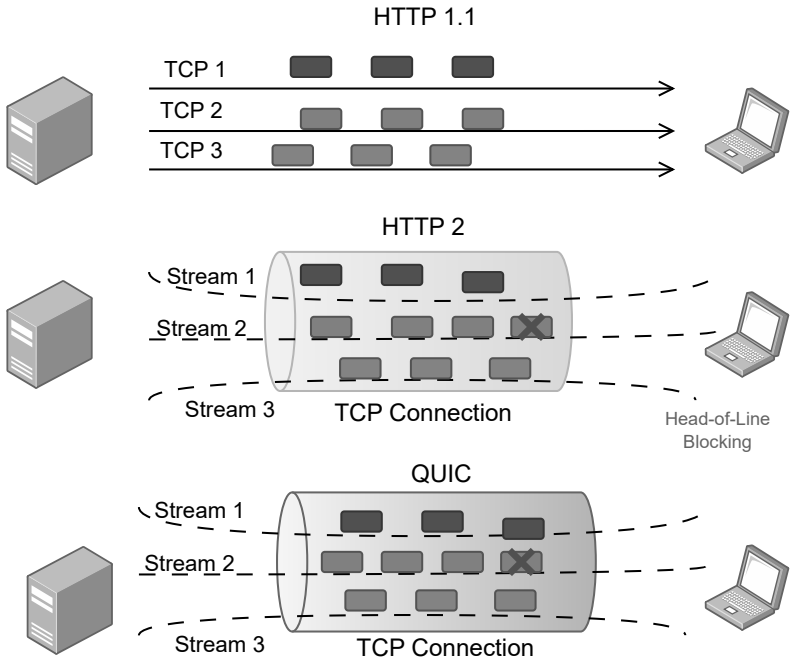

Figure 1.7. *Multiplexing comparison between HTTP1.1 and HTTP/2 over TCP and QUIC (Cui et al. 2017). For a color version of this figure, see www.iste.co.uk/tong/troubleshooting.zip*

HTTP/2 (Figure 1.7(b)) solves this problem by multiplexing multiple data streams over a single connection if there are multiple requests to the server. Data transmitted in the same connection is delivered to the client in order, leading to the TCP HoL blocking. If a TCP packet in a connection is lost, the server must wait until the TCP packet is retransmitted successfully before processing the following TCP packets.

QUIC (Figure 1.7(c)) supports multiplexing multiple data streams over a single connection without requiring ordered delivery of all packets.

Consequently, this can overcome the TCP HoL blocking. Data missing on a stream does not block the delivery of other streams.

1.2.1.4. *Connection migration*

TCP uses a port and an IP address of both endpoints to identify a connection. All network connections will be interrupted when a client changes its IP address such as moving out of a Wi-Fi network range or switching from a wired to a cellular network. However, QUIC uses a connection ID (identifier) randomly generated by a QUIC client to identify a connection. When a QUIC client changes its IP address, it can continue to use the current connection ID from a new IP address without any connection interruptions.

1.2.2. *Other protocols*

1.2.2.1. *IPsec*

Internet Protocol Security (IPsec) (Lopez-Millan et al. 2019) is a secure network protocol that encrypts and authenticates data packets to provide secure communication between two endpoints in the network. IPsec is used widely in virtual private networks (VPN). IPsec is a layer 3 end-to-end security scheme, so it not only protects payload but also IP header of data packets. IPsec uses UDP packets on port 500 for initial handshake, authentication and shared secret establishment. In this protocol, there are two main parts including authentication header (AH) and Encapsulating Security Payload (ESP). In the infancy of IPsec, the former is responsible for authentication while the latter provides data confidentiality. ESP adds a header and a trailer to each packet as in Figure 1.8. IPsec can operate in two modes: transport and tunnel modes. Only data and ESP trailer are encrypted in the transport mode, whereas an entire original packet and ESP trailer are encrypted in the tunnel mode.

1.2.2.2. *TLS*

TLS Rescorla and Dierks (2018) is a successor of Secure Sockets Layer (SSL), a cryptographic protocol designed to provide a communication security in the network. It runs in the application layer to provide privacy and data integrity in data communication. TLS is considered as a secure layer in Hypertext Transfer Protocol Secure (HTTPS), a popular secure communication protocol on the Internet.

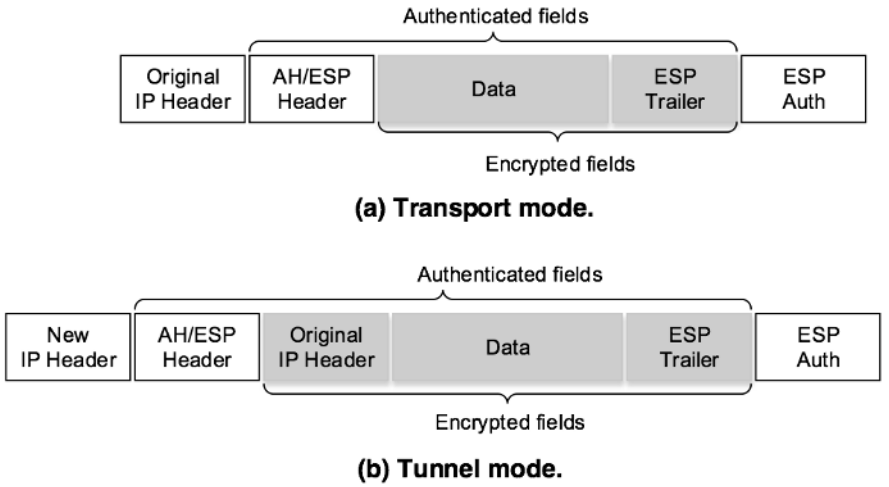

Figure 1.8. *IPsec packet structure. For a color version of this figure, see www.iste.co.uk/tong/troubleshooting.zip*

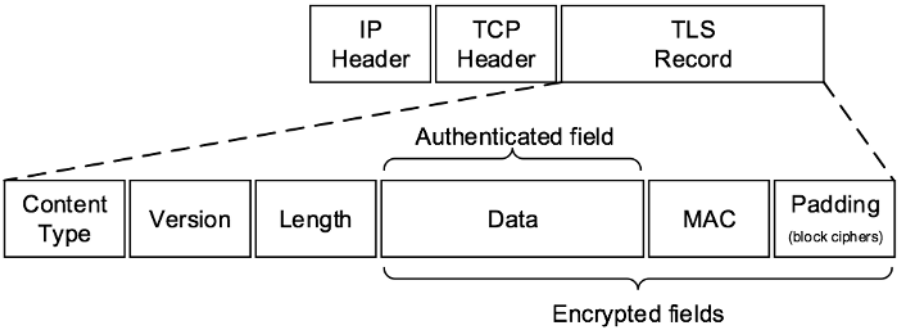

Figure 1.9. *TLS record packet*

TLS contains two main layers including TLS record and TLS handshake. The former one aims to divide data into compressed fragments. In a TLS record, a fragment is followed by message authentication code (MAC). The fragment and its MAC are encrypted together in a TLS record. The structure of TLS record packet is described as in Figure 1.9. During the TLS handshake, clients and servers exchange messages to acknowledge and verify each other. Then, it establishes an encryption algorithm and agrees on session

keys. At the beginning of TLS connections, clients and servers are authenticated according to X.509 certificates chain. In this phase, TLS messages exchanged are not encrypted until session keys are established and agreed. Next, these session keys are utilized by the TLS record to communicate data.

1.3. Drawbacks of troubleshooting with encrypted traffic

Nowadays, many new protocols (e.g. TLS1.3, QUIC, etc.) are designed to provide encryption and authentication for data communication in the network. The authentication protects the data integrity and prevents unexpected modifications in data packets. The encryption provides the confidentiality of transport data and prevents a data inspection for illegal purposes (e.g. stealing information, etc.). This reduces the ability of NOs to obtain an insight into their networks and effectively manage their networks. In this section, we present two drawbacks of network troubleshooting in the context of encrypted traffic.

1.3.1. *Network performance monitoring*

Network performance monitoring is an essential part of the data collection module to collect network parameters for network troubleshooting. Parameter measurement helps NOs to troubleshoot problems and identify whether the problems happen in their network or other sides (e.g. service providers, clients, etc.). Measurement approaches monitor network traffic and use information in packets (e.g. sequence number, acknowledgment number, etc.) to measure network parameters (e.g. quality of experience (QoE), etc.) or additional information (e.g. application class, etc.).

1.3.1.1. *QoE estimation*

Concerning the QoE, many existing studies calculate QoE using a QoE model based on subjective data (Amour et al. 2020). The QoE model is built according to application-level parameters (e.g. start-up delay, number of stalling events, bitrate, resolutions, etc.). These parameters can be extracted by analyzing client or server-side logs, but NOs do not have access to this information. Hence, NOs can implement network-side measurement approaches by analyzing network traffic to obtain these parameters. In fact,

the application-level parameters can be measured thanks to the information in the HTTP packet headers (Schatz et al. 2012; Casas et al. 2014). For example, YouTube's QoE is determined by stalling events in the video playback. Therefore, Schatz et al. (2012) proposed a deep packet inspection technique to measure the number of stalling events using IP network flows. The objective is to measure the playing time accumulated in the buffer of the YouTube player by comparing timestamp of packets and playback time of video frames. When the playback buffer is empty, the video starts to stall. The playback time can be obtained according to the HTTP packet headers. However, service providers (e.g. YouTube, etc.) nowadays encrypt network traffic (e.g. using TLS, QUIC, etc.) to protect user's privacy and data during data transmission. Consequently, QoE estimation approaches using the inspection of HTTP packet headers are no longer available. Therefore, Orsolic et al. (2017) and Orsolic and Skorin-Kapov (2020) proposed a QoE classification approach for YouTube traffic using a machine learning (ML) algorithm. This approach monitors and analyzes network traffic to obtain 54 network features divided into six categories: packet length, transferred data size, packet count, inter-arrival time, throughput and TCP flag count. Then, these parameters are analyzed in the ML algorithm to classify QoE into three QoE classes: low, medium and high QoE. There are many ML algorithms, so this approach evaluates performance of several ML algorithms (OneR, Naive Bayes, SMO, J48 and Random Forest) to select an appropriate one for the QoE classification.

1.3.1.2. *Application identification*

Nowadays, with the development of 5G, there are many strict SLA requirements from end-users (e.g. high reliability, low latency, etc.) (Lin et al. 2019). Consequently, differential treatment is expected by end-users. The emerging of application-aware mechanisms (e.g. network slicing, traffic engineering, etc.) which implement different policies corresponding to various applications, is a potential solution to overcome these drawbacks. In the past, NOs used port numbers or signatures in payload of packets to identify the application classes Lopez-Martin et al. (2017). Libprotoident (Finsterbusch et al. 2013) is an open-source software library that implements the deep packet inspection for protocol identification. Libprotoident only uses port numbers and the first four bytes of payload in each direction. For example, Libprotoident identifies HTTP and SIP flows according to common strings (e.g. ET, PUT, HTTP, SIP, INVI, REGI, etc.) and registered ports

(80, 8080, 8081, 443). Similarly, Adami et al. (2015) implement a deep packet inspection approach to identify the VoIP flows using signatures of SIP protocol (e.g. content-type: "application/sdp", etc.). However, the packet's payload can be encrypted with encryption protocols (e.g. TLS, etc.). Therefore, Deri et al. (2014) proposed nDPI, an open-source library for protocol identification using the information in both header and payload of packets. nDPI can handle TLS-encrypted traffic to identify known applications. In this case, only the network traffic in the initial key exchange can be decoded. nDPI can extract the hostname of servers to identify different applications. For instance, network flows toward the server name "api.facebook.com" are assigned as Facebook application. However, the server name can be encrypted according to DNS over TLS, HTTPS or QUIC Huitema and Rescorla (2018). Consequently, encrypted traffic classification approaches using packet inspection are not effective. This results in many obstacles for the encrypted traffic classification. Amaral et al. (2016) proposed a new traffic classification method using flow-based features and an ML algorithm. This method monitors and analyzes network traffic to extract 12 flow-based features from the first five packets of flows (e.g. packet size, packet time-stamp, inter-arrival time, etc.). Then, these features are analyzed in the ML algorithm to identify the application classes. Lopez-Martin et al. (2017) proposed a traffic classification approach using flow-based features and a deep learning (DL) algorithm to identify 108 application classes for Internet of Things (IoT) network.

1.3.2. *Intrusion detection system*

1.3.2.1. *Firewall*

In the network, middlebox is deployed to block malicious traffic and prevent illegal activities. It can classify network traffic into malicious or benign traffic to make appropriate policies (Kühlewind et al. 2018). In the TLS-encrypted traffic, Service Name Indicator (SNI) is used to identify malicious connections. If SNI in the Client Hello TLS message matches Subject Alternate Name (SAN) in the server certificate, the connection is considered as a benign connection. However, SNI can be encrypted to protect the information about destinations (e.g. domain name of websites, etc.) (Huitema and Rescorla 2018). Nowadays, SNI is encrypted in the Domain Name System (DNS) over TLS, HTTPS and QUIC.

Besides, the middlebox can be used for content filtering (Moriarty and Morton 2018). The middlebox is used to block content to meet requirements from law enforcement or regulatory authorities (e.g. enforcing content-based billing, etc.). They analyze data packets to obtain URL (Uniform Resource Locator) field. If it matches a blocked URL in a blacklist, a 404 error notification is returned. However, these approaches are unavailable when traffic is encrypted. DNS over TLS, HTTPS and QUIC hide the DNS information in the packets.

When network traffic is encrypted, it results in obstacles for malicious traffic prevention and content filtering. If the middlebox cannot inspect network traffic for network management, it needs to be moved to the endpoints at a higher operational cost.

1.3.2.2. *DDoS protection and migration*

Distributed denial-of-service (DDoS) protection is necessary in network operations. It aims to detect DDoS traffic and redirect it to a migration system to filter out DDoS traffic from legitimate traffic.

DDoS can be implemented according to a botnet, a collection of compromised machines or bots (Antonakakis et al. 2012). Many bots nowadays use a domain generation algorithm (DGA) to generate domain names and connect to a Command and Control (C&C) server. Therefore, many studies analyze the domain name of C&C server to detect DGA botnet and prevent DDoS attacks. Woodbridge et al. (2016) proposed a DGA botnet detection system using a long short-term memory (LSTM) network. This system monitors and analyzes DNS packets to obtain the domain name of C&C server. Then, the domain name is analyzed in the LSTM network to detect the DGA botnet. If a domain name is classified as a malicious domain, the IP address of C&C server corresponding to this domain is blocked to prevent DDoS attacks. Tran et al. (2018) proposed LSTM.MI, a LSTM-based DGA botnet detection framework for handling multiclass imbalance. In LSTM.MI, original LSTM is adapted to be cost sensitive to handle the multiclass imbalance. However, DNS information (e.g. domain name, IP address, etc.) is hidden with DNS over TLS, HTTPS and QUIC. This leads to many challenges for the DGA botnet detection systems as well as DDoS prevention.

Besides, DDoS can be detected according to a signature-based intrusion detection system (IDS) (Shurman et al. 2019; Otoum and Nayak 2021). This approach inspects network traffic and checks anomalous signatures in a predefined list of signatures such as file hashes, malicious domains, known byte sequences, etc. For example, Snort AbuHmed et al. (2008) is an open-source IDS for inspecting intruder signatures. Snort rules contain protocol, source IP address, destination IP address, source port, destination port and signatures in the payload for one or more patterns. When network traffic is encrypted, this approach is ineffective. In this case, network features (e.g. packet length, inter-arrival time, etc.) can be used to detect DDoS traffic. Garcia et al. (2021) proposed an anomaly detection system using Artificial Intelligence (AI) for detecting DDoS traffic over encrypted traffic. This approach monitors network traffic and extracts 57 network features (e.g. packet size, number of bytes, etc.). Then, these features are analyzed in a detection model to classify network traffic into legitimate or intrusive. In the detection model, network features are first analyzed in a Gaussian mixture model to calculate a probability density. If it is less than *threshold1*, network traffic is assigned as intrusive. If it is bigger than *threshold2*, it is assigned as legitimate. Otherwise, a reconstruction error is calculated according to Auto-Encoder. If the reconstruction error is less than *threshold3*, it is assigned as legitimate. Otherwise, it is assigned as intrusive. Similarly, Zolotukhin and Hämäläinen (2018) proposed an application-layer DDoS detection method for encrypted traffic. In this method, there are two main phases containing training and detection phases. In the training phase, this method monitors network traffic to collect network features of seven categories: duration of conversation, number of packet sent in 1 s, number of bytes sent in 1 s, average packet size, average TCP window size, average TTL and average of packets with different TCP flags. Then, legitimate traffic is clustered into clusters using an ML algorithm (e.g. K-mean, K-medoids, Fuzzy c-means, etc.). In the detection phase, network traffic is assigned as intrusive when it does not belong to any clusters.

1.4. Conclusion

With the ever-increasing of network devices, computer networks have become more and more complex. Therefore, network troubleshooting plays an important role in network systems (Fonseca and Mota 2017; Yu et al. 2018; Cherrared et al. 2019). This section surveyed related work on several

network problems: rule failure, link failure and switch's buffer overload. Facing a growing number of data theft attacks, many service providers encrypted network traffic to protect data and user's privacy. The network traffic can be encrypted by encryption protocols such as IPsec, TLS, etc. This leads to many obstacles for network troubleshooting for encrypted traffic related to network monitoring approaches and intrusion detection systems.

When the packet's payload is encrypted (e.g. TLS, etc.) or even entire packet is encrypted (e.g. IPsec in the tunnel mode, etc.), it prevents NOs from inspecting network traffic to measure network parameters (e.g. QoE, etc.). Remarkably, the information about the application class is hidden in this case. Consequently, many encrypted traffic classification techniques are studied to identify the applications without decrypting the network traffic. Besides, encrypted traffic results in many difficulties for intrusion detection systems. In the past, IDS detects different kinds of attacks according to its signatures in the network traffic. However, these signatures are hidden due to encrypted traffic. Consequently, IDS today detects different kinds of attacks because of network parameters (e.g. packet length, inter-arrival time, etc.).

The traditional troubleshooting architecture has been studied by the research community over the past decades. Nevertheless, in the context of encrypted traffic, this architecture shows limitations related to network performance monitoring approaches and intrusion detection systems. In the next chapter, we present a novel troubleshooting architecture for encrypted traffic and the data collection module, which aims to collect data for further modules in network troubleshooting.